Dynamické datové struktury IV.

Prioritní fronta.

Tomáš Bayer | bayertom@natur.cuni.cz

Katedra aplikované geoinformatiky a kartografie, Přírodovědecká fakulta UK.

Obsah přednášky

1. Prioritní fronta

Priority Query.

Někdy nazývána “fronta s předbíháním.

Patří mezi zobecněné datové struktury, kombinuje činnost fronty a zásobníku.

Široce používaná struktura pracující s prvky nestejné váhy.

Prioritní fronta představuje strukturu dvojic (klíč,položka) uspořádanou sestupně dle hodnot klíče.

Hodnota klíče (tj. priorita) určena ohodnocovací funkcí, udává “význam” prvku.

Prvek s vyšší prioritou může přeběhnout prvek s nižší prioritou $\Rightarrow$ rozdíl oproti běžné frontě.

Princip prioritní fronty:

Prvky ukládány v pořadí, v jakém jsou přidávány na konci fronty.

Prvky postupně odebírány na základě hodnoty klíče, v každém kroku

2. Operace s prioritní frontou

Implementace prioritní fronty:

- lineární seznam,
- BST,
- halda.

Velké rozdíly ve výkonnostních charakteristikách!

Operace nad prioritní frontou:

- Vytvoření fronty
- Přidání položky: push
- Smazání položky s největší prioritou: pop
- Změna priority položky.
- Výmaz libovolné položky
- Smazání fronty.

3. Ukázka prioritní fronty

Key	Val
204	123
197	239
152	246
142	177
105	432
97	404
92	44
88	164
53	149
45	186
13	98

4. Přehled metod a odhadů složitosti

Metoda	push	pop	find
Neuspořádaný seznam	$O(1)$	$O(N)$	$O(N)$
Uspořádaný seznam	$O(N)$	$O(1)$	$O(1)$
BST, vyvážený*	$O(h)$	$O(h)$	$O(h)$
Heap	$O(\log(N))$	$O(\log(N))$	$O(\log(N))$

*Pro AVL stromy $h = 1.4 \log(N)$, pro degenerované případy BST $h = N - 1$, neefektivní.

5. Implementace PQ neuspořádaným seznamem

Vhodné pouze pro menší datové množiny.

Obousměrný spojový seznam uchováván v neuspořádaném tvaru, značně neefektivní.

Rychlé realizace `push()`, přidání prvku na konec seznamu: $O(1)$.

Velká režie spojená s `pop()`, nutno nalézt nejmenší prvek: $O(N)$!

Drtivá většina operací stejná jako u implementace spojového seznamu.

```
public class Uzel {
  double klic, hodnota;  //par (klic, hodnota)
  Uzel predchozi, dalsi;


  public Uzel (String klic_, hodnota_) { //Konstruktor
      klic=klic_; hodnota = hodnota_;
      predchozi = null; dalsi=null;
  }
}
```

6. Přidání prvku

Operace `push()` stejná jako u běžné fronty:

```
public void push (double klic_, hodnota_){
    Uzel u = new Uzel(klic_, hodnota_);
    if (posledni == null) {  //Prazdna fronta
        prvni = u;
        posledni = u;
    }
    else {  //Neprazdna fronta
        posledni.dalsi = u;  //Nastaveni dalsiho uzlu
        u.predchozi = posledni;  //Nastaveni predchoziho uzl
        posledni = u;  //Nastaveni posledniho uzlu
    }
}
```

7. Odebrání prvku

Operace pop() neefektivní, složitost $O(N)$.

Nepoužitelné pro větší množiny.

Postup:

1. Nutno sekvenčně projít seznam a nalézt prvek $u_{max} = (k_{max}, v)$
   - Pokud je fronta tvořená jedním prvkem, `prvni = null`, `posledni = null`.
2. Prvek u_{max} smazán a přenastaveny pointery:
   - u_{max}`.predch.dalsi =` u_{max}`.dalsi.`
   - u_{max}`.dalsi.predch=`u_{max}`.predch.`
   - u_{max}`=null.`

8. Odebrání prvku, ukázka

```
public void pop() {
  if (prvni.dalsi == null) { //Pouze jeden prvek
      prvni = null; posledni = null;
  }

  else {   //Neprazdna fronta
     Uzel u = prvni, u_max = prvni; //Nalezeni uzlu s max klicem
     for(double k_max = prvni.klic, u = u.dalsi;u!=null;u=u.dalsi ) {
        if (u.klic >k_max) {
           k_max = u.klic;
           u_max = u;
     }
     //Presmerovani odkazu
     u_max_prev.predch.dalsi = u_max_dalsi;
     u_max.dalsi.predchozi = u_max.predchozi;
     u_max = null; //Smazani maxima
  }
}
```

9. Implementace PQ pomocí BST

Výhodnější odhady složitosti pro všechny varianty u vyvážených stromů.

U AVL stromů operace závisí na h, $O(h) = O(1.4 \cdot \log(N))$.

Pro degenerované stromy odhad složitosti $O(N) \Rightarrow$ nutné převažování stromu.

Operace `push()` odpovídá operaci `insert()` v BST.

Modifikace třídy Uzel:

```
public class Uzel {
  int klic, hodnota; //Dvojice (klic, hodnota)
  Uzel levy;   //Odkaz na levy podstrom
  Uzel pravy;  //Odkaz na pravy podstrom

   public Uzel(double klic_, double hodnota_)  { //Konstruktor
      klic = klic_; hodnota = hodnota_;
      levy = null;
      pravy = null;
   }
}
```

10. Odebrání prvku

Složitost $O(1.4 \log(N))$.

Postup využívá pro nalezení uzlu u_{max} cyklus, hledá nejpravější podstrom na nejvyšší úrovni x.

Uzel u_{max} předchůdcem uzlu u, pomocná proměnná p ukládá předchůdce uzlu u_{max}.

Postup:

1. Cyklem nalezení maxima v BST: nejpravější prvek v pravém podstromu.
   - Uzel `u=koren`, u_{max} `=null;`
   - Opakuj dokud `u.pravy != null;`
     - `p =` u_{max}
     - u_{max} `= u`
     - `u = u.dalsi`
2. Smazání uzlu u_{max} s obnovením rovnováhy stromu.

11. Odebrání prvku, ukázka

```
public void pop() {
   Uzel u=koren, umax=null, p = null;
   while (u.pravy != null) {
     p = umax
     umax = u;
     u = u_dalsi;
  }
  //Smaz list
   if ((umax.levy==null)&&(umax.pravy==null))
          smazList(u,p);
  //Smaz uzel 1 potomek
   else if(umax.levy==null||umax.pravy==null)
          smaz1Vetev(umax, p);
  //Smaz uzel 2 potomci
   else smaz2Vetve(umax, p);
}
```

12. Implementace PQ pomocí haldy

Standardní implementace haldy, konstantní složitost operací $O(\log(N))$.

Dosahuje nejlepších výsledků, na rozdíl od BST netřeba strom převažovat.

Využívána datová struktura **maxHeap:**

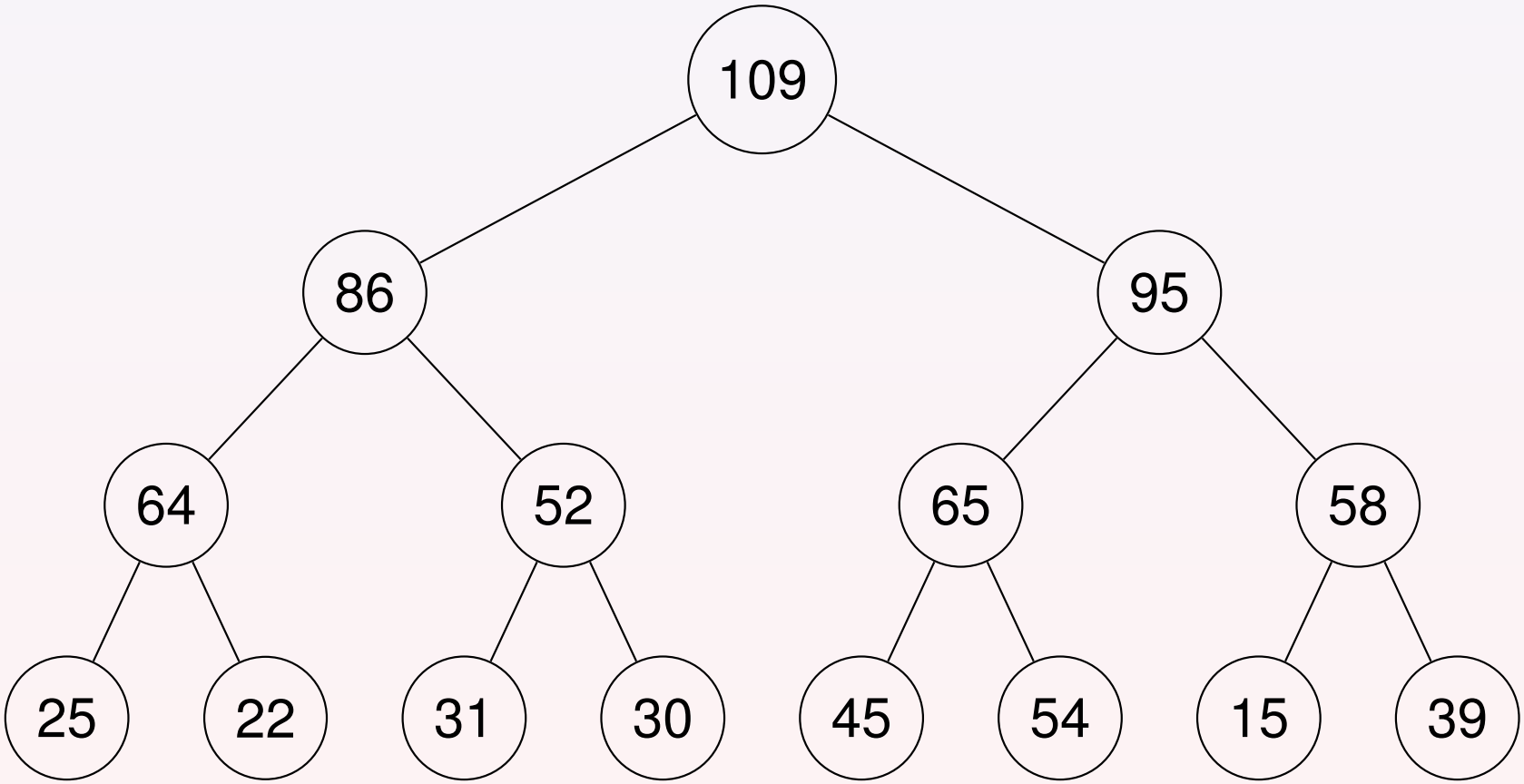
V kořeni maximum, nejmenší hodnoty v listech.

Oproti **minHeapu** u `fixHeapUp()` a `fixHeapDown()` záměna < za >.

Operace `push()`, `pop()` stejné jako u "běžné" haldy.

- `push()`: přidání prvku do haldy.
- `pop()`: odebrání kořene haldy.

13. Ukázka maxHeapu



14. Oprava haldy nahoru

Varianta pro maxHeap

```
public void fixHeapUp(int i) {
   while (i > 1 && (h[i / 2] < h[i])) {  //Zamena > za <
      double temp = h[i / 2];       //Prohozeni s pouzitim
      h[i / 2] = h[i];              //lokalni promenne
      h[i] = temp;
      i = i / 2;                    //Jdi na rodice
   }
}
```

15. Oprava haldy dolů

```
public void fixHeapDown(int k) {
   int i;
   while (2 * k <= n) {
      i = 2 * k;        //Levy potomek
      if (i < n && h[i + 1] > h[i]) { //Porovnani L a P potomka, zamen
         i++;          //Index ukazuje na vetsiho potomka

      if (h[k] < h[i]) //Nesplneny podminky, prohodit. Zamena.
      {
         double temp = h[k];
         h[k] = h[i];
         h[i] = temp;
      } else
         break;     //Ukonceny podminky, nepokracuj

      k = i;          //Pokracuj od prohozeneho potomka
    }
}
```

16. Přidání a odebrání prvku

Přidání prvku do haldy:

```
void push (double item)
{
    n++; //Zvetseni pq o 1
    h[n] = item;
    fixHeapUp(h, n); //Oprav heap nahoru do akt. pozice
}
```

Odebrání kořene haldy:

```
void pop(){
    h[1] = h[h.length()-1];
    fixHeapDown(h, 1); //Oprav heap od korene
    n--;
}
```